GPU-based multi-view rendering

François de Sorbier, Vincent Nozick, Hideo Saito

GPU-Based multi-view rendering

François de Sorbier
Graduate School of Science and Technology
Keio University, Japan
fdesorbi@hvrl.ics.keio.ac.jp

Vincent Nozick
Université Paris-Est
LABINFO-IGM UMR CNRS 8049
France
vnozick@univ-mlv.fr

Hideo Saito
Graduate School of Science and Technology
Keio University, Japan
saito@hvrl.ics.keio.ac.jp

ABSTRACT
Stereoscopic images in computer graphics applications often require two rendering passes reducing by half the frame rate. In this situation, the conversion from standard to stereoscopic images may involve some difficulties to maintain real time rendering if the geometry is made of thousands triangles. Since few years, auto-stereoscopic displays have become more and more popular because of their multi-user capability and because they do not require any specific glasses. However, they usually require five or more input views that can be difficult to generate in real time. In this paper, we present a single pass algorithm using GPU that speeds-up the rendering of stereoscopic and multi-view images. The geometry is duplicated using a shader program that reduces the data transfer between the main memory and the graphic card. It also brings together the computation of some vertices's properties that are similar from one view to another.

Keywords
multi-view, auto-stereoscopy, stereoscopy, real-time, GPU

1. INTRODUCTION
Stereoscopy is a technique that enables to watch three-dimensional images on 2D display thanks, most of the time, to specific glasses. It has many applications in various fields such as data visualization, virtual reality or entertainment because it tends to reproduce our visual perception and makes information easier to understand. In computer graphics, stereoscopic rendering consists in generating two images of a virtual environment from two slightly different viewpoints. In other words, it requires to render the geometry of the scene twice which can double the computational time. In such case, it can be difficult to maintain real-time rendering especially for applications like video-games that are complex in term of geometry and visual effects.

Auto-stereoscopy is a technology recently applied to LCD displays [2] that introduces the ability for one or several users to watch stereoscopic images without wearing any glasses. Depending on their characteristics, auto-stereoscopic displays require from 5 to 64 images [9] to display a single 3D frame. A filter, made of small lenses, is overlaid on the surface of the screen and ensures to emit each image in a specific direction. So, if the user is well located in front of the display, each eye can see a single specific image.

However, the important number of required input images makes it more difficult to maintain real-time rendering compared to a single view rendering. We can state two facts in term of time for standard multi-view rendering. Firstly, data transfer from main memory to the graphic card is costly. Secondly, some operations on vertices remain the same from one view to another which mean there are redundant computations. Global transformations, parts of illumination calculation and texturing are identical for example.

GPU programming is now very popular because it can speed-up many algorithms thanks to an efficient parallelized architecture. Recently, shaders have been updated with a new feature named geometry shader (GS) that takes place between vertex shader and rasterization stages [5]. Geometry shader introduces the possibility to manipulate vertices of input primitives like points, lines or triangles before emitting the result to the rasterization and clipping stages. It becomes also possible to create new primitives during this stage.

The goal of our approach is to exploit geometry shader to speed-up the rendering process of stereoscopic or multi-view images. The ability of geometry shaders to duplicate input primitives allows rendering in a single pass. Multiple sending of the geometry to the graphic card are reduced to a single transfer. Moreover extra computation due to redundant operations is avoided since our algorithm take place after the vertex shader stage.

This paper is structured as follows. We start giving an overview of related and previous works, and then we present a description of our algorithm. In the next section we give details about the implementation of our algorithm with shader codes. Finally, we present and discuss the results of our approach.

2. PREVIOUS WORKS
Several methods have been proposed to overcome the multi-pass rendering limitation for multi-view rendering of 3D information. A point-based rendering solution was proposed by Hubner *et al.* [4] using GPU to compute multi-view splatting, parameterized splat intersections and per-pixel ray-disk intersections in a single-pass. This method reaches 10 fps for 137k points in a 8-view configuration. To increase multi-view rendering performance, Hubner and Pajarola [3] present a direct volume rendering method based on 3D textures with GPU computations to generate multiple views in a single pass. These two solutions significantly decrease the computation time but are not suited for polygon based graphics.

An alternative solution has been proposed by Morvan *et al.* [6] a single 2D image plus a depth map that are interrelated to display multiple views. Although the algorithm reduces the bandwidth of data emitted to the system, it does an assessment over available data to fill the area's missing information of the new views and then reduces the content's truthfulness.

In 2008, de Sorbier *et al.* [7,8] introduced a new single pass algorithm to render stereoscopic and multi-view images using the GPU. This approach is based on a geometry shader implementation and uses multiple rendering target extension (MRT) associated with frame-buffer object (FBO) to save results in distinct textures. Results show that, in some cases, the frame-rate can be twice faster than a multi-pass technique. However, MRT is limited to a single depth buffer shared by all the rendering targets. This restriction is minimized by sorting the triangles in a back to front order that increases computation time. Moreover, hardware constraints limit the number of output textures to eight while some auto-stereoscopic devices require nine viewpoints or more. Finally, it is difficult to integrate this algorithm in an existing application because existing shaders have to be rewritten.

3. ALGORITHM OVERVIEW

Advantages of Shaders

In standard OpenGL implementation, geometry is described as a set of vertices that have to be sent from main memory to the graphic card for each rendering pass. Moreover, there are several function calls and OpenGL state modifications that can affect the frame rate. Obviously, these operations decrease the performance in case of stereoscopic or multi-view rendering because it needs a rendering pass for each view.

By studying the concept of multi-view rendering [2], we can state that some characteristics remain the same from one view to another. Position of vertices is unchanged in the referential of the scene meaning that a transformation matrix is shared over the viewpoints. Likewise, texture coordinates, light vector and normal of a vertex are identical. So, each characteristic independent of the viewpoint might be computed once to increase the performances. Then, each vertex should be duplicated according to a given viewpoint.

In that sense, shaders provide useful functionalities to merge some operations and duplicate only what is necessary. A vertex shader is designed to apply several independent processing on each vertex while a geometry shader is dedicated to handle primitives. In particular, this GPU stage allows creating or removing vertices, to emit new primitives and to apply transformations and takes place just after the vertex shader.

Single Texture Based Approach

Previous work [8] on GPU-based multi-view rendering demonstrates that it is possible to considerably speed-up the process. However, the algorithm is difficult to use because of constraints like the lack of an efficient depth test or the use of multiple textures during the rendering stage.

Our approach takes advantage of a single texture to make possible the use of the depth buffer. The goal is to generate all the views and to render it at the correct position. Moreover, with only one texture we can significantly increase the number of views that was previously reduced at eight for a single rendering pass because of hardware limitations. Obviously, it means that the size of the texture is proportional to the number of views and the size of the original view.

In our case, the different views are organized over the whole texture in sub-areas. Each of them has a resolution of (w,h). For example, six views will spread over six sub-areas $\mathrm{SA}(i,j)$ where $0 \le i < 3$ and $0 \le j < 1$ and the texture resolution is $(3\times w, 2\times h)$. Furthermore, clipping stage takes place after the geometry shader. According to these facts, we have to consider transformations to correctly project the geometry in the texture and apply one more step to refine clipping.

Geometry Transformation

Input triangles of the geometry shader are projected on an area that covers the whole texture. So we have to apply operations on these triangles in order to transform them to fit the bottom-left sub-area of the texture. Then, triangles are duplicated and moved over their assigned sub-area.

Since all the input views share a common image plan, the transformation on triangles is a 2D operation composed of a scaling S and a translation T that have to be applied after the projection P of vertices. If the distribution of the views on the texture is defined as $\mathrm{NV}(x,y)$, then we define S as $S = 1/\mathrm{NV}(x,y)$. Since the projection is normalized, we defined T as:

$$T = -1 + 1/NV(x,y)$$

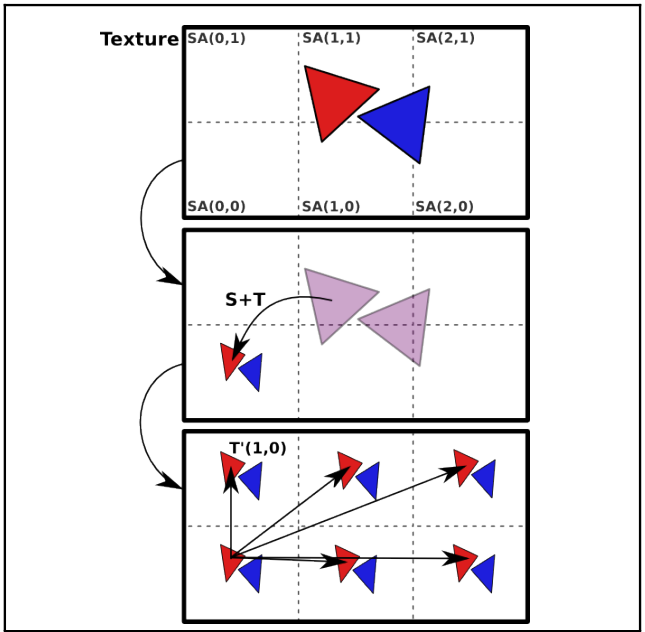

Figure 1. Description of the transformations required to duplicate the geometry using a single texture.

To send a duplicated triangle in a specific sub-area $\mathrm{SA}(i,j)$, we apply one more translation T' defined as:

$$T'(i,j) = 1/\mathrm{NV}(i,j) \times 2 \times (i,j)$$

The full process is depicted in Figure 1.

The result of these transformations is that there is no difference between the views in each sub-areas. So, we need to apply one more operation to take into account the perspective transformation. For multi-view rendering, the only required parameter is the eye separation Δ that defines the distance between two adjacent viewpoints. Since projection volumes are unchanged and view directions are parallels, we define the

stereoscopic transformation as a translation T_s perpendicular to the view direction. T_s can be applied after using matrix M that transforms each vertex into camera space. It means that the translation is $T_s(i,j) = (\Delta \times (i + j \times nbv), 0, 0)$ where $\mathbf{nbv}$ is the number of sub-area in a row.

Geometry transformation can be summed up as:

$$\text{Vertex}_{SA}(i,j) = T'(i,j) + P' \times (T_s(i,j) + M \times Vertex)$$

(Equation 1.)

with $P' = T \times S \times P$

Clipping

Using a single texture means that the clipping will be applied on the border of the texture and not on each sub-area borders. Geometry transformation described in previous sub-section, will uncover new triangles in incorrect sub-area that will not disappear after clipping (Figure 2).

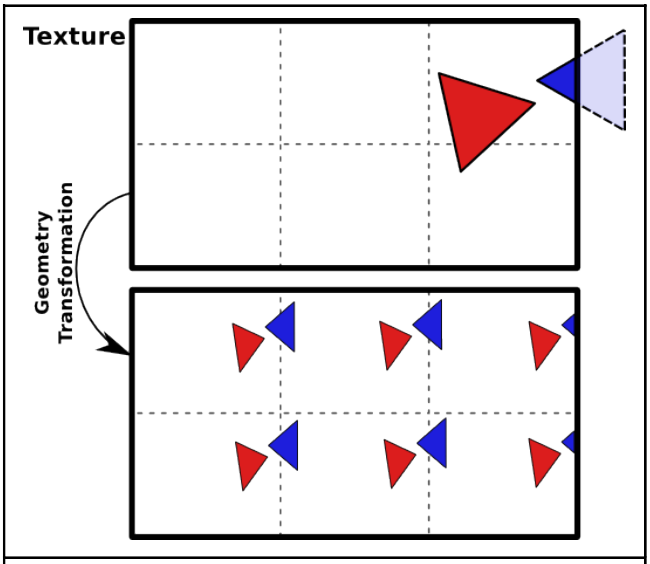

Figure 2. Description of clipping constraints when the different views are shared on a single texture.

To partially or completely hide incorrect triangles, we need to apply our own clipping. It exists three options: do the clipping in the geometry shader, in the fragment shader or in both.

In the geometry shader, triangles outside or intersecting a border of a sub-area are detected using the Cohen-Sutherland method. Then the triangle is emitted, discarded or clipped depending on the result. However, the clipping requires to compute intersections with the border and, if necessary, to create new triangles.

In the fragment shader, fragments that are outside of their sub-areas are discarded. However, it requires testing a large number of fragments.

The third solution is to clip triangles using both shaders. The geometry shader detects and discards triangles that are outside of the sub-area. Then the number of fragments to test in fragment shader is reduced.

Advantages and limits of each approach are compared in the discussion section.

4. IMPLEMENTATION

This section describes the implementation of our algorithm using OpenGL 2.1 and GLSL 1.2. The result of our method is saved in a texture using the *Frame Buffer Object* extension.

The Vertex Shader

The goal of the vertex shader is to centralize the common operations from one view to another one. Equation 1 shows that the transformation matrix M (MODELVIEW matrix) is similar for each vertex. So each vertex can be multiplied with that matrix during the vertex shader stage.

```
#version 120


void main(void){
gl_Position = gl_ModelViewMatrix * gl_Vertex;
}
```

Figure 3. One possible code for the vertex shader

Other parameters can be computed in vertex shader like texture coordinates, light vectors depending of the normal, color of vertices.

The Geometry Shader

```
#version 120

#extension GL_EXT_geometry_shader4 : enable

flat varying ivec2 subarea;  // Coordinates of a sub-area
uniform vec2 screensplit;    // Subareas distribution on texture
uniform int numberofviews;
uniform float eyesep;        // Eye separation

void main(void){
  vec2 T = -1.0+1.0/screensplit;
  mat4x4 TSP = mat4x4(1.0/screensplit.x,0.0,0.0,0.0,
                      0.0,1.0/screensplit.y,0.0,0.0,
                      0.0,0.0,1.0,0.0,
                      T.x,T.y, 0.0,1.0)*gl_ProjectionMatrix;
  float start = -float(numberofviews*0.5)*eyesep;
  if(mod(numberofviews,2)==0) start += eyesep*0.5;
  for(int current=0;current<numberofviews;++current){
    subarea.y = int(floor(current/screensplit.x));
    subarea.x = current%int(screensplit.x);
    for(int i = 0 ; i < 3; ++i){  // for each point of the triangle
      vec4 tmp = TSP*(gl_PositionIn[i]+vec4(start,vec3(0.0));
      gl_Position = tmp;
      gl_Position.xy += subarea/screensplit*tmp.w*2.0;
      EmitVertex();
    }
    EndPrimitive();
    start += eyesep;
  }
}
```

Figure 4. One possible code for the geometry shader

In the geometry shader, we apply the transformations presented in the previous section. For each sub-area, vertices of the input triangles are duplicated and translated according to the corresponding viewpoint. The result is multiplied with the OpenGL projection matrix, translated, and scaled to fit the sub-areas. All this operations must in homogeneous coordinates to correspond with OpenGL matrix.

We add an extra variable *subarea* that is used to associate a triangle with its sub area coordinates. This is useful to easily identify a fragment in the fragment shader step and be able to compute the clipping.

The code in Figure 4 presents our approach with the clipping performed only in the fragment shader. The clipping based on geometry and fragment shaders needs only few changes. It checks if triangles are completely outside the sub-area boundaries whether or not. The test is done by computing the position of each vertex when they are transformed into sub-area $SA(0,0)$. If at least one vertex is in this sub-area then we continue the process else we discard the triangle. It slightly reduces operations in geometry shader and the number of emitted fragments.

We will omit the implementation details of the clipping with geometry shaders, since performances of this approach are quite bad compared to the two others.

The Fragment Shader

The traditional fragment shader is increased by a single test for clipping. Each fragment is associated with its target sub-area thanks to the variable *subarea*. With this information and the size of a sub-area, it is possible to know if the fragment has to be discarded or not.

```
#version 120

uniform ivec2 subareasize;

flat varying ivec2 subarea;

void main(void){
      ivec2 coord = ivec2( gl_FragCoord.xy ) – subarea *
subareasize;

   if( coord.x > screensize.x || coord.y > screensize.y ||
       coord.x < 0 || coord.y < 0){
      discard;
   }else
      gl_FragColor = vec4(1.0,0.0,0.0,1.0);
   }
}
```

Figure 5. One possible code for the pixel shader

If the fragment is kept then any kind of operation can be applied on it like illumination per pixel (Figure 6, Figure 11, Figure 12), texturing. So there are only few modifications in the fragment shader code.

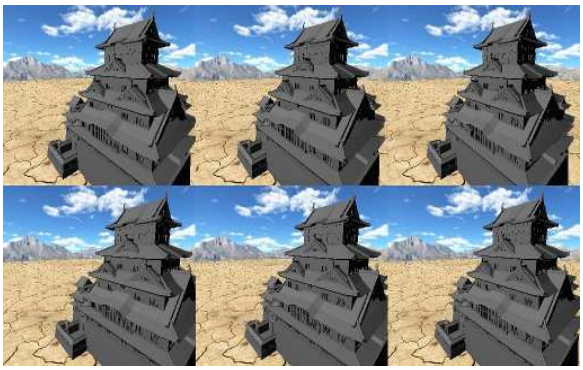

Figure 6. Himeji Castle, 6 views. 35,200 triangles. Lighting per pixel.

5. RESULTS & DISCUSSIONS

Performances Analysis

We experimented our algorithm on a *bi-Xeon 2,5Ghz* running Linux. The graphic card is a *nVIDIA GeForce GTX 285* with *1Go* memory. The algorithm was tested using different kind of models with various numbers of triangles and graphical effects. The resolution of each view is 1024×768 . No special data structure like VBO was used.

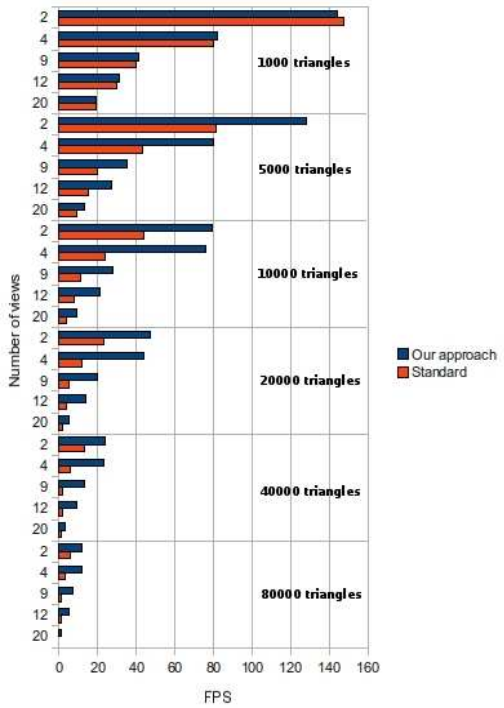

Figure 7. Performances of our approach compared to the standard one. Tests are applied with various numbers of views and triangles.

Figure 7 presents the performances obtained using our approach compared to the standard multi-pass rendering. We evaluate the

results over different number of triangles and views. If the scene is made of one thousand triangles then we notice that performances of our algorithm are similar or, in case of two views, slightly worst compare to the normal one.

In all other analyzed conditions, performances of our multi-view rendering are around twice better than the standard multi-view rendering. Rendering with four view-points shows that our results can be three time better for more than 5000 triangles. Especially in this case, the differences between two and four views are small.

Performances are closely dependent of the number of input primitives. For low number of triangles, our approach is less effective than the standard one because the number of OpenGL drawing calls does not exceed the transfer capabilities from the main memory to the graphic card.

Results are similar for two and four views. So, in Figure 8, we analyze our rendering algorithm with 1 to 17 view-points and 10000 triangles. In the first fourth cases, performances are quite similar then after, an important drop in frame-rate occurs until 13 view-points. Finally, performances seem to become stable again. We think that under a given amount of data, geometry shaders can parallelize operations but they will become a bottleneck in the other case.

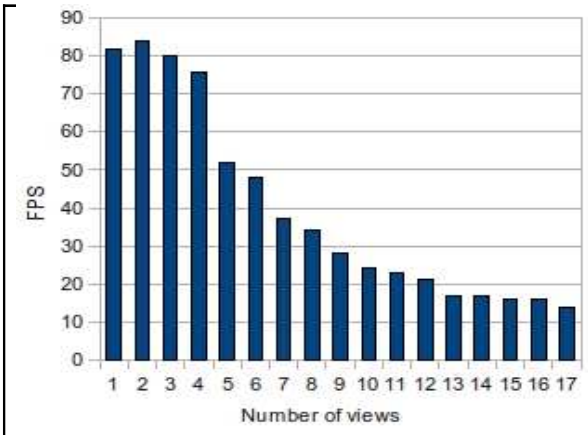

Figure 8. Frame-rate of our approach on a scene with 10,000 triangles in function of the number of views.

The frame-rate for one view is less than the one for two views. This is because our algorithm apply some operations that are useful for multi-view rendering but make no sense for a single view.

We also wanted to compare the performances between clipping only in the geometry shader (GSC) and clipping in the geometry and fragment shaders (GFSC). Results are presented in Figure 9. using different number of views and triangles.

We observe that with 10000 triangles, frame-rate applying GFSC is better than GSC. This difference is more important with four views which tends to confirm our analysis of behavior of geometry shader. With 40000 triangles GSC have similar results than GFSC independently from the number of views. Number of operations in geometry shaders becomes too much important and leads to a bottleneck. Actually, pixel clipping takes advantage of the *unified shader architecture* of modern GPU that allow to reallocated a pool of processors to a specific shader which also explain why results are quite similar.

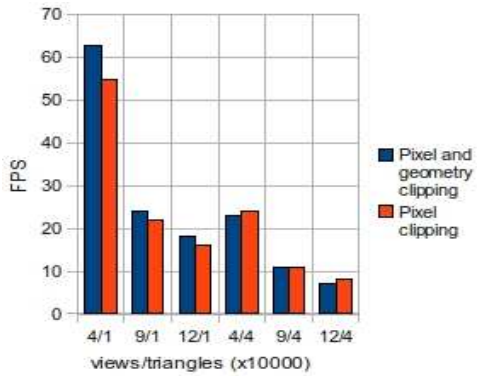

Figure 9. Comparison of the frame rate between a clipping in pixel shader stage and a clipping in geometry and pixel shader stages.

1.1. Gaming Scenario

We applied our approach on a real case to demonstrate that we can easily integrate it in an existing implementation. We used *Nexuiz* [1] which is an open-source *First Person Shooter* (*FPS*) game.

Figure 10 shows that we can obtain 34 frames per second while rendering four views. Note that in, Figure 10, the 2D interface remains of single view.

However, the performances were not as high as expected. First, video games tend to reduce the number of triangles which is in line with the explanations of the previous sub-section.

Second, video games use data structures such as VBO to quickly send data to the graphic card. We explain in the following sub-section the consequence of this.

Figure 10. *Nexuiz*, 4 views

OpenGL 3

At this time, we only used our algorithm with the OpenGL *glVertex* function to send the geometry at the graphic card. We tested our approach with some data structures like *Vertex Buffer Objects* (*VBO*) that copy directly the data on the graphic card, but results of our algorithm were lower than a standard multi-view rendering.

In previous works of de Sorbier *et al.*[8], they were using *VBO* and could obtained promising results. At this time, they were using an *nVIDIA geForce GTS 8600* graphic card with OpenGL 2.1 which is older than ours.

Our opinion is that the architecture of modern GPU has significantly been improved and is now dedicated to deal with new requirements of OpenGL 3. It means that our implementation have to fit the same requirements to take advantage of the capabilities of such a graphic card.

6. CONCLUSIONS

We have presented an algorithm to generate stereoscopic and multi-view images using the GPU in a single pass. We take advantage of the geometry shader to speed-up the rendering process by duplicating the geometry on the graphic card and avoiding redundant computations. Our algorithm overcomes the limitations introduced in previous works, such as shared depth buffer and restrictions on the number of output-views. Moreover, the implementation of this approach is now simplified.

We generate all the views on a single texture which requires only one depth buffer. We explained the transformations applied on the input triangles to duplicate and spread them over the single texture. We also introduced different solutions to resolve the clipping problems when a duplicated triangle overlaps two views.

The results showed that performances of our approach vary according to the number of triangles and views. The algorithm is efficient when it processes more than 1000 triangles otherwise, benefits of the geometry shaders are under-exploited. We also noticed that the frame-rate is quite similar to render two, three or four views but decreases while rendering more views because geometry shader becomes a bottleneck. But our approach always remains better than the standard multi-pass rendering. Results are two times higher or even three times for example in case of four views rendering. Finally, we compared our solutions for the clipping problem and explained that performances are similar mainly because the *unified shader architecture* of modern GPU.

We presented an implementation of our approach in a game but performances were reduced because of the requirement of OpenGL 3 on recent graphic cards. So, our next goal is to update our algorithm to be able to use data structures like VBO using the new design of OpenGL 3.

7. ACKNOWLEDGMENTS

Part of the work presented in this paper was supported by the FY2009 Postdoctoral Fellowship for Foreign Researchers from the Japan Society for Promotion of Science (JSPS) and by the National Institute of Information and Communications Technology (NICT).

8. REFERENCES

Alientrap, Nexuiz, http://www.nexuiz.com, 2009.

Dodgson, N. A. 2005. Autostereoscopic 3d displays. Computer. 38(8):31–36.

Hubner, T. and Pajarola, R. 2007. Single-pass multi-view volume rendering. In Proceedings of International Conference Computer Graphics and Visualization.

Hubner, T., Zhang, Y. and R. Pajarola, R. 2006. Multi-view point splatting. In GRAPHITE'06, 285–294.

Lichtenbelt, B. and Brown, P. 2007. EXT gpu shader4 Extensions Specifications. NVIDIA.

Morvan, Y., Farin, D., and de With, P. H. N. 2007. Joint depth texture bit-allocation for multi-view video compression. In Picture Coding Symposium (PCS).

de Sorbier, F. Nozick, V. and Biri, V. 2008. Accelerated stereoscopic rendering using gpu. In 16th International Conference in Central Europe on Computer Graphics, Visualization and computer Vision'2008 (WSCG'08), ISBN 978-80-86943-16-9, February 2008.

de Sorbier, F, Nozick, V. and Biri, V. 2008. GPU rendering for autostereoscopic displays. In 4th International symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08), June 2008.

Takaki, Y 2006. High-density directional display for generating natural three-dimensional images. In Proceedings of the IEEE, volume 94, 654–663.

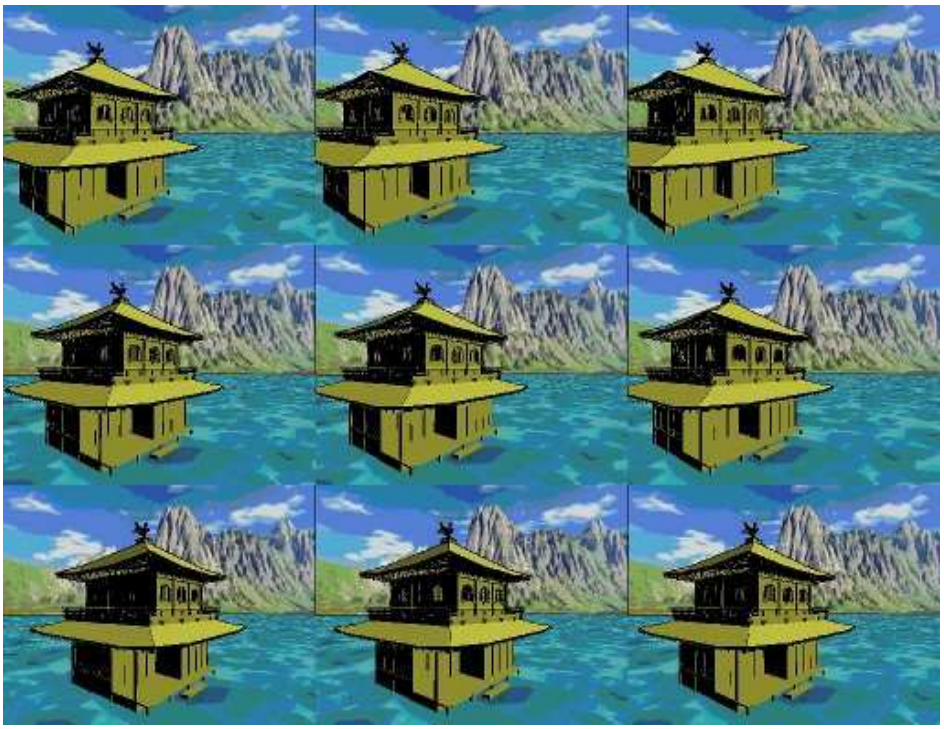

Figure 11. Kyoto Golden Pavilion, 9 views. 23,400 triangles. Toon shading effect with borders emphasis.

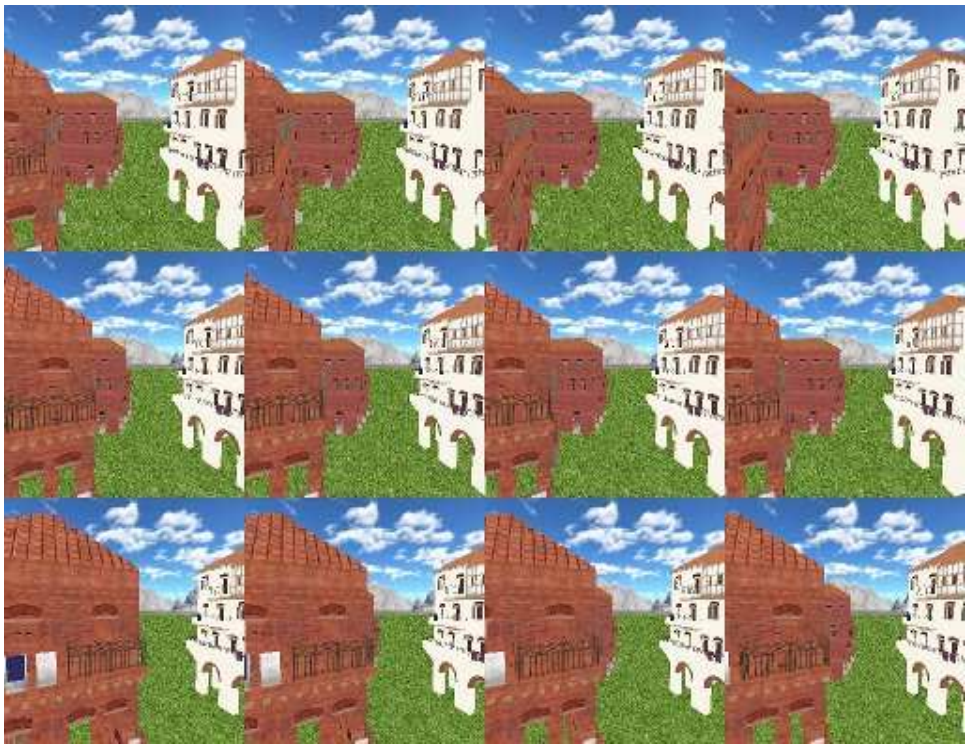

Figure 12. Rome from *City Engine* (procedural.com), 12 views. 86,300 triangles. Texturing with shaders.